Prototyping Processes

L. S. Barbosa

Abstract

Construction and observation are two basic notions in Computer Science corresponding to precise dual mathematical concepts: those of algebra and coalgebra. This paper introduces a simple coalgebraic model for concurrent processes and discusses its animation in the declarative language CHARITY. It is argued that the ability to reason in an uniform way about data and behaviour, provides an unifying approach to functional prototyping of software specifications.

Keywords: Coalgebraic models, prototyping, higher-order programming.

1 Introduction

The success of formal approaches to software engineering depends a great deal on the interplay between suitable notions of *specification*, *prototyping* and *refinement*.

Given a *formal specification* notation, prototyping allows for a stepwise development style. Each design stage being immediately animated, quick feedback about its behavior is gathered within the design team. *Refinement*, on the other hand, stands for the process of calculating implementations from specifications over concrete architectural platforms.

Functional programming languages have been a favorite vehicle for rapid prototyping of formal specifications, at least since Peter Henderson's me too [13] proposal for animating VDM [19]. Several animation tools have emerged, both in academia an industry, such as RAISE [9], VDM-SL [7], B TOOL [21] and CAMILA [2]. The direct use of modern functional languages supporting rich type systems, such as HASKELL [16] or STANDARD ML [12] in system's modeling has also been advocated.

However, the increasing demand for distributed (even mobile) applications is challenging the application of formal methods, entailing the need to scale up this picture. In such systems, both information and computational power are disseminated along a (eventually loose) network of autonomous and interacting agents. The key ingredient to control — in terms of which specifications are stated — becomes the *observational pattern* of the components involved and their possible combinations. Models which, being non isomorphic at the data level, behave in a similar way "as far as we can see", tend to be identified, in practice.

Dep. Informática, Univ. Minho, Portugal. E-mail: lsb@.di.uminho.pt

Work partially supported by LOGCOMP under PRAXIS XXI contract 2/2.1/TIT/1658/95.

The work reported here is part of a research agenda on extending the *specification - prototyping - refinement* trilogy to the development of concurrent and distributed systems. Such systems can be seen as transition structures — or *coalgebras* [26] — whose shape is determined by a signature of *observers* and *actions* on a (hidden) state space. The concept is dual to that of an (inductive) data structure, such as sequences or generalized trees, which emerges canonically as a term algebra for a signature of *constructors*. In fact, the intuitive symmetry between construction and observation (data and behaviour) is mirrored, semantically, on the algebra *vs* coalgebra duality.

In this context, the paper discusses how a prototyping kernel for processes can be built on top of a language which, being functional in style, provides the necessary categorical constructions to define both inductive and coinductive datatypes, *i.e.*, canonical models for algebras and coalgebras. CHARITY [5] is such a language.

Section 2 provides a brief introduction to CHARITY and, at the same time, discusses how the familiar non inductive structures of sets and maps, widely used in specifications, can be defined. Then, in section 3, more elaborated coinductive types are introduced as prototypes of processes. An example is discussed and a fragment of a process algebra defined. Section 4 claims that the refinement of processes' state spaces may induce richer behavioural patterns and shows how this development can be animated in CHARITY. Finally, some possible generalizations are briefly mentioned.

2 Prototyping in Charity

Unlike more traditional functional languages, CHARITY is not based on the λ-*calculus*. Its basis is the term logic of distributive categories enriched with a definitional mechanism for datatypes [6]. This means, in particular, that function composition, instead of function application, is taken as the fundamental primitive in the language. Moreover, the basic building blocks of CHARITY programs are combinators arising as universal arrows in category theory.

What makes CHARITY an interesting alternative for prototyping purposes, if compared with other declarative languages, is the very general way it provides for defining datatypes as *algebras* or *coalgebras* for functors.

It is well known that a specification signature, *i.e.*, the formal analog to a software component interface, can be expressed by an endofunctor on an appropriate category, stressing the fact that operations act uniformly both on types and type morphisms. Given one such functor T, a T-algebra is simply an arrow from $\mathsf{T}S$ to a S, S being said the *carrier* of the algebra. Therefore, an algebra specifies how values in the carrier are built from its constructors.

This simple notion is extremely expressive. In particular, there is a one to one correspondence between algebras for certain Set-functors and the usual notion of an algebra in Universal Algebra, Set denoting the familiar category of sets and set-theoretical functions. By reversing the arrow, one gets the dual notion of a coalgebra: an arrow from S to $\mathsf{T}S$ which specifies how values of the carrier, now thought of as

a *state space*, are observed or updated by the operations in T [1].

Consider, to begin with, the CHARITY declaration of the natural numbers, defined inductively by the two familiar constructors.

```
data nat -> C = zero: 1 -> C
              | succ: C -> C.
```

Its purpose is twofold. On the one hand it declares the signature of the natural numbers, in the form of a functor $\mathsf{T}X = 1 + X$, 1 standing for the final object in the category ("the" singleton set, if Set is assumed). Simultaneously, it provides a canonical algebra for T, *i.e.*, an arrow from $\mathsf{T}W$ to the set W of terms, which is initial in the sense that every other T-algebra factorizes through it. In general, it is the existence and uniqueness of such an arrow that makes possible definition and proof by induction, respectively. Such datatypes are then classified as *inductive*.

From a declaration like `nat` above one gets for free a case expression (determined by the `nat` constructors, `zero` and `succ`), a map construction, *i.e.* the action of T on functions, and a general fold operation. For an arbitrary algebra $\alpha : TS \longrightarrow S$, the last is known as a *catamorphism*, written as $(\![\alpha]\!)_{\mathsf{T}}$. It is nothing more than the computer science nick name for the universal arrow from the initial algebra to α.

This kind of inductive, tree-like structures, are the functional programmer favorite stuff. In the formal methods community, however, the data modeling primitive selected as "first choice", when facing a design problem, would most probably be some kind of *map*, expressing a functional dependence. This is a subtle, but expressive, shift of perspective.

In fact, such unordered structures, like maps or sets, are easier to observe than to construct (in an effective computational sense). Of course, they have a more or less obvious implementation in functional languages as certain kinds of sequences. The prototype of a map in CAMILA, for example, is a sequence of pairs. Operations over maps are translated to operations over sequences. We will look here for more direct representations.

2.1 Powerset

Our starting point is to represent sets by their characteristic functions. Note that, however, the exponential type (the type B^A of functions from a type A to B) is not primitive in CHARITY. This is because the underlying categorical model is not cartesian closed, just distributive, and therefore functions are not values. The way to introduce function spaces resorts to the possibility of defining higher-order types, as a particular case of coinductive datatypes [27]. Let us see how.

The definition mechanism for *coinductive* datatypes is similar to the one provided for the inductive case. A specification is given of the different ways in which such a type can be observed, but nothing is said about how to construct its values. Formally a coinductive datatype for a signature functor T is a final T-coalgebra, *i.e.*, a transition structure of shape T over the space of all possible observations.

[1] The reader is referred to [18] for an introduction to the use of algebras and coalgebras in computing.

In this context, consider the following representation for $\mathbf{2}^A$, the powerset of A.

```
data C -> set(A) = in: C -> A => bool.
```

Here a set is seen as a structure accessed, or observed, by a predicate (the observer `in`) codifying set membership: all one is able to know about a set is whether a particular value is in it. Technically, this amounts to saying that `set(A)` is the final coalgebra for the constant functor $\mathsf{T}X = \mathbf{2}^A$. A coalgebra is just an arrow from a type X to $\mathsf{T}X$. Being final means it is unique (or universal) among all such arrows. Of course, $\mathsf{T}X$ being a constant functor, the state space for $\mathsf{T}X$-coalgebras becomes irrelevant.

Simple operations on sets are defined as operations on predicates, as in, for example, the following representation of the empty set, set union and difference.

```
def empty : 1 -> set(A)
    = () => (in: x => false).

def union: set(A) * set(A) -> set(A)
    = (s1,s2) => (in: x => or(in(x,s1),in(x,s2))).

def diff: set(A) * set(A) -> set(A)
    = (s1,s2) => (in: x => and(in(x,s1), not in(x,s2))).
```

As the datatype `set(A)` is parametric in `A`, some operations will require a specific definition of equality on `A`, as is the case of singleton set formation.

```
def sing{equal: A * A -> bool}: A -> set(A)
    = a => (in: x => equal(a,x)).
```

Such parameterization is straightforward in CHARITY. For example, ZF abstraction is just written as,

```
def zf{pred: A -> bool}: set(A) -> set(A)
    = s  => (in: x => and (in(x,s), pred(x)) ).
```

Finally, `set{f}`, for any function `f` from `A` to `B`, denotes the datatype action on morphisms [2].

There are however some familiar operations over sets that can not be programmed in this way as they rely on (the axiom of) choice. Computationally this means that they presuppose the existence of an ordered representation of a universe with respect to which the sets of interest are defined. It is typically the case of set equality, subset inclusion and monoidal reductions: all of them require the ability to pick an element from a set.

[2] Technically, `set(-)` is the *co*-type functor for the bifunctor $(A, X) \mapsto \mathbf{2}^A$ [8].

Our solution to this problem consists in isolating the choice dependent operations and providing a separate specification of a universe and a choice function over it. Sequences offer a simple implementation, but in CHARITY one is not limited to finite structures. Therefore, a typical encoding of choice is made over possible infinite streams. They are known as *colists* and defined coinductively by

```
data C -> colist(A) = delist: C -> SF(A * C).
```

where `SF(X)` is the exception construction, *i.e.*, the initial algebra for $\mathsf{T}X \cong 1 + X$. Having defined the universe `U(A)` as `colist(A)`, two choice functions are provided, the second accepting an additional filter predicate [3].

```
def choice: set(A) * U(A) -> SF(A)
    = (s, u) =>  col_first_st{ x => in(x,s) } u.

def pchoice{pred}: set(A) * U(A) -> SF(A)
    = (s, u) =>  col_first_st{ x => and(in(x,s), pred(x)) } u.
```


```
def choice: set(A) * U(A) -> SF(A)
    = (s, u) =>  col_first_st{ x => in(x,s) } u.

def pchoice{pred}: set(A) * U(A) -> SF(A)
    = (s, u) =>  col_first_st{ x => and(in(x,s), pred(x)) } u.
```


2.2 Maps

A direct implementation of *maps*, also as a coinductive datatype, follows almost the same lines. The space of maps from A to B, written as $A \rightharpoonup B$, is defined as

```
data C -> maps(A, B) = ap: C -> A => SF(B).
```

stating that elements of $A \rightharpoonup B$ are observed through evaluation (the observer `ap`, for "apply"), which may return an undefined value. The identity mapping is defined by a case expression and composition expressed in terms of the exception type values,

```
def mid: set(A)  -> maps(A, A)
    =  s  =>  (ap: a =>  { true => ss(a)
                         | false => ff
                         } in(a,s)  ).

def mcomp: maps(A, B) * maps(B, C) -> maps(A, C)
    = ((ap: t), (ap: r))  => (ap: a => compose_SF{t,r} a).
```

[3] The `SF(-)` type is extensively used in the sequel. Its constructors are concretely named `ff` : $\mathbf{1} \longrightarrow$ `SF(-)` and `ss` : `SF(-)` $\longrightarrow$ `SF(-)`. The function `col_first_st` returns the first element of a colist satifying a given predicate.

Typical operations over maps, such as the ones in the meta-language of VDM, are easily supported. For example, consider the following definitions of overwrite and domain restriction.

```
def over: maps(A,B) * maps(A,B) -> maps(A,B)
    = (m1,m2) => (ap: x => { ff    => ap(x,m1)
                           | _     => ap(x,m2)
                           } ap(x,m2)  ).

def dr: maps(A,B) * set(A)  -> maps(A,B)
    = (m,s) =>  (ap: x => { true  => ap(x,m)
                          | false => ff
                          } in(x,s)  ).
```

Also notice that, being parametric in two arguments, its action on morphisms is divided in three cases: acting on the domain through `f`, `maps{f,x => x}`, on the range through `g`, `maps{x => x,g}`, or both, `maps{f,g}`.

In traditional formal specification methods one is used to restrict herself to *finite* maps. Such restriction, however is not essential in our CHARITY implementation. The following is an example of an infinite map which maps every even natural number to its successor.

```
def mm: 1 -> maps(nat,nat)
    = () => (ap: n => {true => ss(succ n) | false => ff} even(n) ).
```

3 From Data to Processes

3.1 Processes

The last example above suggests that coinductive types are closer to the notion of a *process*, as a computational entity with a proper state space persisting in time, than to the idea of a data container.

To illustrate how processes can be dealt with in CHARITY, consider the specification of a (simplified fragment of a) bank account management system (referred to as *Bams* in the sequel) starting from the following signature:

$$
\begin{array}{rl}
bal : Id \times X \longrightarrow Am & \text{(show balance)} \\
hol : Id \times X \longrightarrow \mathbf{2}^{Ho} & \text{(show set of account holders)} \\
cre : Id \times Am \times X \longrightarrow X & \text{(credit)} \\
deb : Id \times Am \times X \longrightarrow X & \text{(debit)} \\
aho : Id \times Ho \times X \longrightarrow X & \text{(add account holder)} \\
rho : Id \times Ho \times X \longrightarrow X & \text{(remove account holder)}
\end{array}
$$

where Am, Id and Ho stand for the types modelling amounts, account identifiers and account holders, respectively. X models the state space of the *Bams* system.

Notice that the two first operations are *pure observers*, usually called *attributes* in the object-orientation literature, both parameterized by the account identifier. On the other hand, the last four are *actions* (or *methods*) which change the state space without any immediately visible effect. This information may be collected into a functor. Take the following as a first approximation.

$$\mathsf{T}X = Am^{Id} \times (\mathbf{2}^{Ho})^{Id} \times X^{Id \times Am + Id \times Am + Id \times Ho + Id \times Ho} \quad (1)$$

$$= Am^{Id} \times (\mathbf{2}^{Ho})^{Id} \times X^{Id \times (Am + Am + Ho + Ho)} \quad (2)$$

Note that the main operator in the definition of T is a *product*, whereas in a functor arising from an algebraic signature *coproducts* are usually taken. In fact, in the latter case one is concerned with the different available ways to build values. In this case, however, the possibility of (pure) observations being carried in parallel couldn't be discarded. Also observers and actions can be activated simultaneously, eventually by different processes, as they are non interfering operations.

In fact, one may argue that the system can process either the first attribute, the second one or both simultaneously. So, let us replace $Am^{Id} \times (\mathbf{2}^{Ho})^{Id}$ by $(Am \times \mathbf{2}^{Ho})^{Id \amalg Id}$, where $Id \amalg Id$ is defined by the coproduct $Id + Id + Id \times Id$. The construction $A \amalg B$ behaves like a partial product (or an extended coproduct) and is so common that we decided to include it in the prototyping kernel.

```
data amg(A, B) -> X = c0:  A -> X
                    | c1:  B -> X
                    | c01: A * B -> X.
```

Another enhancement consists in introducing partiality. In fact, it is most unlikely that the above mentioned operations can be modeled by total functions. Observers are clearly partial operations. This amounts to take as their output type the exception coproduct, modeled by `SF(-)`, as explained above. T takes now the form

$$\mathsf{T}X = ((Am + 1) \times (\mathbf{2}^{Ho} + 1))^{Id \amalg Id} \times X^{Id \times (Am + Am + Ho + Ho)} \quad (3)$$

Actions could also be partial. However, partiality in this case should be absorbed by the identity function over the state space. This procedure caters for a distinction between the usual data partiality and dynamic partiality, seen as the possibility of termination, which arises by replacing X by $X + 1$. This leads us to the following general shape for coalgebras modelling this kind of processes:

$$\gamma : S \longrightarrow B^C \times S^A \quad (4)$$

where A has the form of a coproduct (of action arguments) and B as well as C are usually a $\amalg$ construction. This is defined in CHARITY as

```
data S -> obj(A, B, C) = ob : S -> C => B
                       | ac : S -> A => S.
```

In this setting, the bank system is specified by giving a concrete representation for the state space and the operations. From the above signature for *Bams* one arrives to

```
data BT =  obj(Id * coprod(Am, Am, Ho, Ho),
               SF(Am) * SF(set(Ho)),
               amg(Id,Id))
```

As a particular instance of this process type consider a coalgebra over a state space defined as a map from account identifiers to the respective balance and set of holders, *i.e.*,

```
def BSt = maps(Id, Am * set(Ho)).
```

The process itself is introduced as a constant:

```
def bams: 1 -> BT
 = () =>
 (| s  => ob : x => { c0  i      => (SFp0 ap(i,s), ff)
                    | c1  j      => (ff, SFp1 ap(j,s))
                    | c01 (i, j) => (SFp0 ap(i,s), SFp1 ap(j,s))
                    } x
  |       ac : (i,x) =>
               { s40 a => cre(s,(i, a))
               | s41 a => deb(s,(i, a))
               | s42 h => aHo(s,(i, h))
               | s43 h => rHo(s,(i, h))
               } x
  |) bseed.
```

The code for the observers is straightforward: just apply the right argument to the right component of the state space. Note the possibility of observing the balance and the set of holders simultaneously, either for the same account or not. As the result of observing, say, the balance of a particular account may return undefined (if, for example, the account does not exist), a way is needed to either record this fact or project the required information from the tuple. This is the purpose of functions `SPp0` and `SPp1` [4].

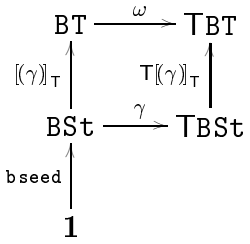
Note that the actual process `bams` is a *concrete* coalgebra over a `BSt` variable. But what can be prototyped is only its *behaviour*, *i.e.*, its canonical image in the final coalgebra. This kind of abstract process universes is exactly what CHARITY has to offer. Therefore `bams` is introduced as an *anamorphism* [8] for a concrete coalgebra γ, whose code is written between the `(|` and `|)` brackets.

Anamorphims are the formal duals to catamorphisms, corresponding to the notion of *unfolding* in functional programming. In fact, the state of a process is just the type

[4] The actual definition is, *e.g.*, `def SFp0:  SF(A * B) -> SF(A) = ff => ff | ss(p) => ss p0(p)`. In the `bams` definition, `s4i` denotes the ith embedding in a 4 sumands coproduct.

over which unfold proceeds. Technically, given a T-coalgebra γ, the anamorphism $[\![\gamma]\!]_{\mathsf{T}}$ is the unique arrow from γ to the final T-coalgebra.

Finally, the anamorphism is applied to a *seed* value, `bseed`, specifying an initial value for the state space. The following diagram summarizes what's going on,



where ω is the final coalgebra which emerges from the CHARITY declaration.

The actual way in which the anamorphism is computed (and the process image revealed) resorts to lazy evaluation. In each action activation a new continuation process is returned upon which experimentation proceeds.

3.2 A Process Algebra

How do processes get composed? Algebraic, stateless software components are joined together by functional composition, originating tree-like modular structures. The composition patterns for processes, on the other hand, are considerably richer. Firstly, the internal state of each process being composed cannot be discarded. Secondly, interaction among them proceeds during the overall computation. The result is a network of processes joined by the matching between the input and output parameters in their interface. The kind of connectives relevant are essentially variants of parallel composition, restriction and renaming found in *process algebras* [22, 15, 14].

Instead of defining such combinators on the fly, our approach begins by organizing processes — understood as concrete coalgebras for some kinds of polynomial functors — in suitable categories. Then, the relevant compositional patterns emerge as canonical constructions on the category, rather than being fixed by intuition. Such connectives have correspondents in different processes categories (*e.g.*, arising from different families of functors), providing a remarkable level of genericity.

From a methodological point of view this approach is in debt to Abramsky's *interaction categories* [1]. Our processes, however, are concretely defined over (also) concrete state spaces, leading to a rather different structure. In particular, by the presence of such state spaces, most diagrams, including the ones expressing identity and associativity of composition, only commute up to isomorphism. In consequence, we end up with a bicategorical structure, the overall approach being much closer to what has been proposed by R. Walters and his team [20].

Presenting in detail the bicategory Pr_{T}, in which processes like the *Bams* example live, is out of the scope of this paper (the interested reader is referred to [3]). Our intention here is just to present CHARITY codifications of some of the relevant connectives.

Such connectives are, again, defined as anamorphisms. For example, the following corresponds to the synchronous execution of processes `p` and `q` — notice how the result interface is expanded.

```
def osyn: obj(A1, B1, C1) * obj(A2, B2, C2)
              -> obj(A1 * A2, B1 * B2, C1 * C2)
    = o =>
      (| (p,q) => ob: (x1, x2) => (ob(x1, p), ob(x2, q))
       |          ac: (a1, a2) => (ac(a1, p), ac(a2, q))
       |)  o.
```

The corresponding abstract operator is a tensor product in Pr_T. Another tensor in Pr_T, giving a less strict interpretation of parallel composition, is `opar`. The intuition is that, putting `p` and `q` side by side results in an (observable) increase of behaviour: not only the individual observers and actions of both processes are available, but there is also the possibility of activating them concurrently (the disjointness of the two state spaces avoiding interference).

```
def opar: obj(A1, B1, C1) * obj(A2, B2, C2)
            -> obj(amg(A1,A2), amg(B1,B2), amg(C1,C2))
    = o =>
      (| (p,q) => ob: x => { c0 c => c0 ob(c, p)
                           | c1 d => c1 ob(d, q)
                           | c01 (c,d) => c01 (ob(c, p), ob(d, q))
                           } x
       |          ac: x => { c0 a => (ac(a, p), q)
                           | c1 b => (p, ac(b, q))
                           | c01 (a,b) => (ac(a, p), ac(b, q))
                           } x
       |)  o.
```

Interaction between two processes is achieved by connecting output to input gates. For the kind of processes discussed here, of T shape, output is only produced by observers. With respect to input, however, there are two possibilities: the values generated by, say, process `p` can be supplied to `q` either as attribute parameters or as input to `q` actions.

In the former case they are used to trigger `q` observers. The resulting process becomes a coalgebra for

$$\mathsf{T}_{12}(X) \;=\; D^C \times X^{A_1 \amalg A_2}$$

for $\mathsf{T}_1(Y) = B^C \times Y^{A_1}$ and $\mathsf{T}_2(Z) = D^B \times Z^{A_2}$, the shapes of `p` and `q`, respectively. In the latter case, by contrast, `p` output is an argument of a `q` action, C becoming an action in the composed process. The interface functor is then

$$\mathsf{T}_{21}(X) \;=\; D^{C_2} \times X^{A \amalg C_1}$$

for $\mathsf{T}_1(Y) = B^{C_1} \times Y^A$ and $\mathsf{T}_2(Z) = D^{C_2} \times Z^B$, respectively. We call the resulting combinators `ohook` and `ahook`, as their effect is to send a hook from the first to the second process, activating, respectively, observers or actions.

```
def ohook: obj(A1, B, C) * obj(A2, D, B) -> obj(amg(A1,A2), D, C)
    = o =>
      (| (p,q) => ob: c => ob(ob(c, p), q)
       |          ac: x => { c0 a => (ac(a, p), q)
                            | c1 a' => (p, ac(a', q))
                            | c01 (a,a') => (ac(a, p), ac(a', q))
                            } x
       |)  o.


def ahook: obj(A, B, C1) * obj(B, D, C2) -> obj(amg(A,C1), D, C2)
    = o =>
      (| (p,q) => ob: c => ob(c, q)
       |          ac: x => { c0 a => (ac(a, p), q)
                            | c1 c => (p, ac(ob(c, p), q))
                            | c01 (a,c) => (ac(a, p), ac(ob(c, p), q))
                            } x
       |)  o.
```

Other process operators we have considered include restriction and renaming, both particular cases of pre- and post- composition with a function; coproduct, which corresponds to what is known as external choice in some *process algebras*; replication and general feedback.

All of them are first defined in Pr_{T}. Then their CHARITY codifications as operators on *final* (rather than arbitrary) T-coalgebras are calculated in a way such that a particular diagram is made to commute. Such a diagram states that the operator is uniform in the universe of T-coalgebras. A sufficient, but not necessary, condition for this arises whenever the operator is induced by a natural transformation between the origin and target T shapes [3].

4 Animating a Refinement

The main stream of our research is directed towards modelling interactive processes by concrete coalgebras over Set and building a refinement calculus in this setting. Such a calculus helps not only in deriving distributed implementations of software systems in a mathematically sound way but also in identifying procedural redundancy in existing systems [3].

A particular sub-problem we have been studying is the effect of static refinement of state spaces in the overall process dynamics. At this level we resort to the SETS calculus [24] for static data refinement. Refinement means, in this framework, an epimorphic transformation between sets introducing data redundancy, in a controlled (*i.e.*, recoverable) way, in order to achieve greater conformity with a given implementation platform. This transformation presumably entails efficiency.

A SETS law is an (in)equation of the form $A \trianglelefteq_f^{\phi} B$, stating that every instance of A can be reified into the corresponding instance of B, by adopting abstraction epimorphism f and provided that concrete invariant ϕ is enforced over B. On

the whole, and using the terminology of [23], the following abstraction invariant is synthesized: $\lambda ab\ .\ (a = f(b)) \wedge \phi(b)$.

One such law concerns map partition. For example, the *Bams* state space can be factorized into two maps: one dealing with account balances and the other with holders' information management. More precisely we get

$$Id \rightharpoonup Am \times \mathbf{2}^{Ho} \ \trianglelefteq\ \ (Id \rightharpoonup Am) \times (Id \rightharpoonup \mathbf{2}^{Ho}) \tag{5}$$

a refinement witnessed by a surjection — the *abstraction function* — which joins the two maps into one.

Such a state transformation leads to a possible distribution of the state space into two independent processes. Moreover, independent actions can be carried out in parallel.

Once calculated, such a refinement can be effectively prototyped in CHARITY. The two development stages can even coexist as prototypes and the formal relation between them — the abstraction function — also turned into executable code.

Without further comments, we present below this development step for the *Bams* example, since it is almost self-explanatory. We begin with the type declarations and then define the processes dealing with the two halves of the *Bams* system.

```
data ASt = maps(int, int).
data HSt = maps(int, set(string)).

data AT = obj(int * sum2(int, int), SF(int), int).
data HT = obj(int * sum2(string, string), SF(set(string)), int).

data BT1 = obj(amg(int * sum2(int, int), int * sum2(string, string)),
               amg(SF(int), SF(set(string))),
               amg(int, int)).

def AcTab: 1 -> AT
 = () =>
 (| s  => ob : i => ap(i,s)
  |       ac : (i,x) =>
               { s20 a => cre(s,(i, a))
               | s21 a => deb(s,(i, a))
               } x
  |) atex.



def HoTab: 1 -> HT
 = () =>
 (| s  => ob : i => ap(i,s)
  |       ac : (i,x) =>
               { s20 h => aHo(s,(i, h))
               | s21 h => rHo(s,(i, h))
               } x
  |) htex.
```

Finally, the new *Bams* system is formed by the parallel composition of `AcTab` and `HoTab`:

```
def bdt': 1  -> BT1  = () => opar(AcTab, HoTab).
```

5 Concluding

Different models for processes, seen as concrete coalgebras for different Set-functors, may be easily modeled in CHARITY. For each of them, a variety of composition patterns emerge from the (bi)categorical structure by universality, giving rise to connectives which may also be animated in the prototyping kernel. Besides the ones used above, some other typical examples include:

- `data X -> proc(A, B) = pr:  X -> A => B * X`, *i.e.*, total deterministic processes, also known as Mealy automata (*cf.*, [20]).
- `data X -> parp(A, B) = pp:  X -> A => SF(B * X)`, *i.e.*, partial processes.
- `data X -> objt(A, B) = ob:  X -> B * A => X`, *i.e.*, Moore automata, a particular simple model for objects.
- `data X -> repr(A, B) = rr:  X -> A => set(B * X)`, *i.e.*, non deterministic processes, extending binary relations in time.
- `data X -> prc(A, B, C, D) = o :  X -> C => D | a :  X -> A => (B * X)`, a generalization of the model considered in this paper, such that actions may also produce observable output.
- `data X -> prc(A, B, C, D) = o :  X -> C => D | a :  X -> A => (SF(B * X))`, the partial, possibly terminating, extension of the previous one.

In summary, we have tried to show how a sufficiently strong prototyping kernel for processes can be based on CHARITY, providing the same kind of functionality one is used to require from traditional prototyping languages. This makes it possible to reason in an uniform way about data and behaviour, the algebraic and coalgebraic aspects of specifications.

This kind of categorical modelling of datatypes, either in the constructive, algebraic side, or in the behavioural, coalgebraic one, was initiated in Hagino's landmark thesis [11], and has become, since then, a major influence in the design and calculation of algorithms [4]. In fact, programming exclusively in terms of generic functionals directly derived from datatype definitions, such as catamorphisms or anamorphisms, leads to a controlled, data driven, use of recursion. This may be as beneficial to declarative programming as the removing of `goto` statements has been to imperative languages twenty years ago. The use of coinductive types, as recursive types inhabited by infinite objects, to model reactive systems has also been studied in type theoretic frameworks (*e.g.*, [10]).

Finally, note that, in the approach sketched here, the canonical model of a process is a coalgebraic structure over its observation patterns. Taking such patterns

systematically has been the overall concern of the work in *process algebras* for the last two decades. This has been carried to an extent that discards the actual observed data, actions and observers being identified as symbols in a formal language. What may distinguish the approach taken here, however, is the explicit association of a transformational contents to actions. A similar use of coalgebraic structures has been made, by [17] and [25], in the semantics and specification of object oriented systems, although in a more axiomatic style.

References

[1] S. Abramsky, S. Gay, and R. Nagarajan. Interaction categories and the foundation of typed concurrent programming. In M. Broy, editor, *Deductive Program Design: Proc. of the 1994 Marktoberdorf Summer School*. NATO ASI Series F, Springer Verlag, 1994.

[2] J. J. Almeida, L. S. Barbosa, F. L. Neves, and J. N. Oliveira. CAMILA: Prototyping and refinement of constructive specifications. In M. Johnson, editor, *6th Int. Conf. Algebraic Methods and Software Technology (AMAST)*, pages 554–559, Sydney, December 1997. Springer Lect. Notes Comp. Sci. (1349).

[3] L. S. Barbosa. A coalgebraic approach to process refinement. In *Proc. 14th International Workshop on Algebraic Development Techniques (WADT'99)*, Bonas, France, 15-18 September 1999. (in print).

[4] R. Bird and Moor. O. *The Algebra of Programming*. Series in Computer Science. Prentice-Hall International, 1997.

[5] Robin Cockett and Tom Fukushima. About Charity. Yellow Series Report No. 92/480/18, Dep. Computer Science, University of Calgary, June 1992.

[6] Robin Cockett and Dwight Spencer. Strong categorical datatypes II: A term logic for categorical programming. *Theor. Comp. Sci.*, (139):69–113, 1995.

[7] J. Fitzgerald and P. G. Larsen. *Modelling Systems: Pratical Tools and Techniques in Software Development*. Cambridge University Press, 1998.

[8] M.M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, University of Twente, Dept INF, Enschede, The Netherlands, 1992.

[9] C. George. The RAISE specification language: a tutorial. In *Proc. of VDM'91*. LNCS (551), 1991.

[10] E. Giménez. Co-inductive types in COQ: An experiment with the alternating bit protocol. Tr 95-38, INRIA Rocquencourt-CNRS-ENS Lyon, June 1995.

[11] T. Hagino. Category theoretic approach to data types. Ph.D. thesis. Technical Report ECS-LFCS-87-38, Laboratory for Foundations of Computer Science, University of Edinburgh, UK, 1987.

[12] R. Harper and K. Mitchell. Introduction to standard ML. Technical Report, University of Edimburgh, 1986.

[13] P. Henderson. me too: A language for software specification and model building. Preliminary Report, University of Stirling, 1984.

[14] M. C. Hennessy. *Algebraic Theory of Processes.* Series in the Foundations of Computing. MIT Press, 1988.

[15] C. A. R Hoare. *Communicating Sequential Processes.* Series in Computer Science. Prentice-Hall International, 1985.

[16] P. Hudak, S. L. Peyton Jones, and P. Wadler. Report on the programming language Haskell, a non-strict purely-functional programming language, version 1.2. *SIGPLAN Notices*, 27(5), May 1992.

[17] B. Jacobs. Objects and classes, co-algebraically. In C. Lengauer B. Freitag, C.B. Jones and H.-J. Schek, editors, *Object-Orientation with Parallelism and Persistence*, pages 83–103. Kluwer Acad. Publ., 1996.

[18] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–159, 1997.

[19] Cliff B. Jones. *Software Development — a Rigorous Approach.* Series in Computer Science. Prentice-Hall International, 1980.

[20] P. Katis, N. Sabadini, and R. Walters. Bicategories of processes. *Journal of Pure and Applied Algebra*, 115(2):141–178, 1997.

[21] K. Lano. *The B Language and Method: A Guide to Practical Formal Development.* FACIT. Springer-Verlag, 1996.

[22] A. J. R. G. Milner. *Communication and Concurrency.* Series in Computer Science. Prentice-Hall International, 1989.

[23] C. Morgan. *Programming from Specification.* Series in Computer Science. Prentice-Hall International, 1990.

[24] J. N. Oliveira. A reification calculus for model-oriented software specification. *Formal Aspects of Computing*, 2(1):1–23, 1990.

[25] H. Reichel. An approach to object semantics based on terminal co-algebras. *Math. Struc. Comp. Sci.*, (5):129–152, 1995.

[26] J. Rutten. Universal coalgebra: A theory of systems. Technical Report CS-R9636, CWI, Amsterdam, 1996.

[27] M. A. Schroeder. Higher-order Charity. Master's thesis, The University of Calgary, 1997.